



1995-04

Finite Element Approximation of Large Air Pollution Problems I: Advection

Neta, Beny

<http://hdl.handle.net/10945/39463>



Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>

**FINITE ELEMENT
APPROXIMATION OF LARGE AIR
POLLUTION PROBLEMS I:
ADVECTION**

Francis X. Giraldo
NRC Research Associate
Naval Postgraduate School
Department of Mathematics
Monterey, CA 93943

Beny Neta
Naval Postgraduate School
Department of Mathematics
Code MA/Nd
Monterey, CA 93943

19 April 1995

Abstract

An Eulerian and semi-Lagrangian finite element methods for the solution of the two dimensional advection equation were developed. Bilinear rectangular elements were used. Linear stability analysis of the method is given.

1. Introduction

Two photographs appearing in the New York Times (March 28, 1994) show the damage of air pollution near big emission sources. But the problem exists even away from sources since air pollutants can be transported, mainly by advection. Thus air pollution becomes a global problem. This physical phenomenon consists of three major stages (see e.g. Zlatev [1]):

1. emission,
2. transport/advection,
3. transformation during the transport which includes: diffusion, deposition and chemical reactions.

In this paper, we only discuss the transport stage and the solution of the two dimensional advection equation by finite element methods.

2. Finite Element Solution

The two dimensional advection equation is given by

$$\frac{\partial c}{\partial t} = -\frac{\partial}{\partial x}(uc) - \frac{\partial}{\partial y}(vc), \quad x_L \leq x \leq x_R, \quad y_L \leq y \leq y_U, \quad 0 < t \leq T \quad (1)$$

where c is the concentration of a certain pollutant and u and v are the wind velocity components in the x and y directions, respectively. Clearly, when one is interested in several pollutants, the equation is replaced by a system of such equations coupled only via the chemical interaction between species.

The methods for numerical solution of the advection equation can be divided into five groups:

1. Finite differences,
2. Spectral methods,
3. Finite volume,
4. Characteristic-based methods or semi-Lagrangian,
5. Finite elements.

The finite difference methods are most popular and have been analyzed thoroughly (see e.g. Richtmeyer and Morton [2]). Spectral methods (see e.g. Orszag [3,4]) are used in weather forecasting, but not very much in air pollution. The pseudo spectral methods are of the same group. Here the solution is approximated by a truncated polynomial whose derivatives are substituted in the equation. The spectral methods require periodic boundary conditions. Finite volume or cell method is based on the integral form of the equation. The computational domain is divided into elements (volumes or cells) within which the integration is carried out. This method preserves the property of conservation (see Peyret and Taylor [11]). The semi-Lagrangian methods are not very popular among scientists working with air pollution models, but these are now gaining popularity in weather prediction. "Discretization schemes based on a semi-Lagrangian treatment of advection have elicited considerable interest... since they offer the promise of allowing larger time steps (with no loss of accuracy) than Eulerian-based schemes whose time step length is overly limited by consideration of stability" (see Staniforth and Côté [9]). Semi-Lagrangian methods based on finite difference or finite element spatial discretization were developed.

Here we discuss the finite element approximation to the two dimensional advection equation. Both Eulerian and semi-Lagrangian finite elements will be discussed and tested. Software will be available upon request or electronically via world wide web at the URL address <http://math.nps.navy.mil/~bncta>. The advantage of finite elements is the fact that the discretization can be as easily carried out for nonuniform grids. Thus one can use a fine grid only where the action is and a coarser grid away from there. First order linear one dimensional elements have been previously used, see e.g. Pepper et al [5]. We now discuss bilinear finite elements on rectangles. It was shown by Neta and Williams [6] that isosceles triangles with linear basis functions and rectangular bilinear elements are superior to other triangulations and to finite differences. If the grid is uniform, rectangular elements are preferred since Staniforth et al [7] have shown how to evaluate the integrals efficiently and the mass matrix can be replaced by a tensor product of two tridiagonal matrices. If the grid is nonuniform again the rectangular elements are preferred, since the isosceles triangles lose their shape.

3. Bilinear Finite Elements

Discretize the rectangular domain, by introducing the nodes

$$(x_i, y_j), \quad i = 0, 1, \dots, I+1, \quad j = 0, 1, \dots, J+1,$$

where

$$x_0 = x_L, x_{I+1} = x_R, y_0 = y_L, y_{J+1} = y_U. \quad (2)$$

Suppose we number the interior nodes

$$n = 1, \dots, IJ \quad (3)$$

from bottom left to top right, see figure 3 for the case $I = 4, J = 3$.

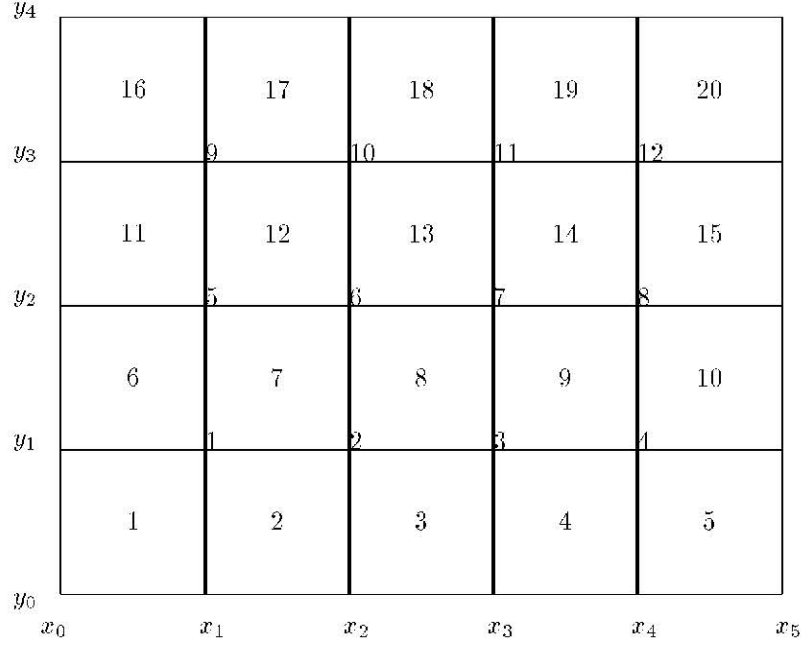


Figure 3: node and element numbering

The number of finite (rectangular) elements is $N_e = (I + 1)(J + 1) = 20$ in this case.

We now define the basis functions $\varphi_m(x, y)$ as bilinear functions on each rectangle, so that

$$\varphi_m(x, y) = \begin{cases} 1 & \text{at node } m \\ 0 & \text{at all other nodes.} \end{cases} \quad (4)$$

To obtain the bilinear basis functions defined on the k^{th} element, we can make a transformation of this rectangle to a square centered at the origin having sides of length 2 (see figure 4).

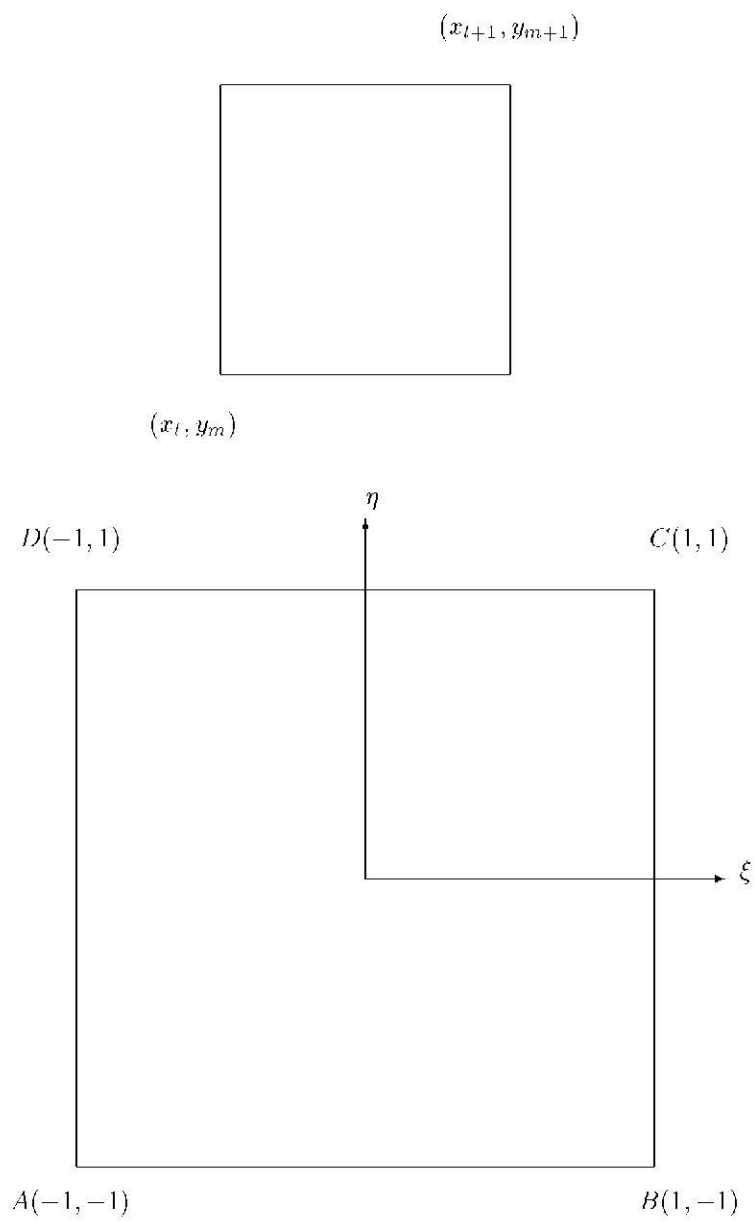


Figure 4: k^{th} element (top) and its transformed one

The transformation is given by

$$\begin{aligned}\xi &= \frac{2}{x_{l+1}-x_l}x - \frac{x_{l+1}+x_l}{x_{l+1}-x_l} \\ \eta &= \frac{2}{y_{m+1}-y_m}y - \frac{y_{m+1}+y_m}{y_{m+1}-y_m}\end{aligned}\tag{5}$$

and the basis functions in the $\xi - \eta$ domain are given by

$$\begin{aligned}\varphi_A &= \frac{1}{4}(\xi - 1)(\eta - 1) \\ \varphi_B &= -\frac{1}{4}(\xi + 1)(\eta - 1) \\ \varphi_C &= \frac{1}{4}(\xi + 1)(\eta + 1) \\ \varphi_D &= -\frac{1}{4}(\xi - 1)(\eta + 1)\end{aligned}\tag{6}$$

where the subscripts denote the vertex at which $\varphi = 1$. Note that the basis functions are product of the appropriate linear basis functions, i.e.

$$\varphi_A(x, y) = e_l(x)e_m(y)$$

where

$$e_i(\theta) = \frac{\theta_{i+1} - \theta}{\theta_{i+1} - \theta_i}.$$

This property is crucial to efficiently evaluating the integrals (Staniforth et al [7]).

The approximate problem becomes

$$M\dot{c} - Kc = b\tag{7}$$

where the entries of the matrices M , and K are given by

$$M_{ij} = \int \int_R \varphi_j \varphi_i dx dy\tag{8}$$

$$K_{ij} = \int \int_R \left(u \varphi_j \frac{\partial \varphi_i}{\partial x} + v \varphi_j \frac{\partial \varphi_i}{\partial y} \right) dx dy.\tag{9}$$

The vector c gives the concentrations at grid points at any time t , and b gives the boundary data

$$b_j = - \sum_{i=1}^{N_e} c_i(t) \left[\int_{y_L}^{y_U} (u \varphi_i \varphi_j) \Big|_{x_L}^{x_R} dy + \int_{x_L}^{x_R} (v \varphi_i \varphi_j) \Big|_{y_L}^{y_U} dx \right]\tag{10}$$

Since u, v are in general functions of x and y , we use numerical quadrature to evaluate K_{ij} . The quadrature we employed in our case is the two point open type, i.e.

$$\int_a^b f(x) dx = \frac{3h}{2} [f(a+h) + f(a+2h)],\tag{11}$$

where

$$h = \frac{b-a}{3}$$

and the error term is given by

$$\frac{3}{4}h^2 f''(\xi).$$

Thus for the first integral in K_{ij} we get

$$\int \int_R u \varphi_j \frac{\partial \varphi_i}{\partial x} dx dy = \sum_{k=1}^{N_e} \int_{x_l}^{x_{l+1}} \int_{y_m}^{y_{m+1}} u \varphi_j \frac{\partial \varphi_i}{\partial x} dx dy. \quad (12)$$

Now use the quadrature for each integral and centered differences for the partial derivatives to get

$$\begin{aligned} \sum_{k=1}^{N_e} \frac{3}{2} h_x \frac{3}{2} h_y \Big\{ & u(E) \varphi_j(E) \frac{\varphi_i(P) - \varphi_i(Q)}{2\delta} + \\ & u(H) \varphi_j(H) \frac{\varphi_i(Y) - \varphi_i(Z)}{2\delta} + \\ & u(F) \varphi_j(F) \frac{\varphi_i(V) - \varphi_i(S)}{2\delta} + \\ & u(G) \varphi_j(G) \frac{\varphi_i(W) - \varphi_i(X)}{2\delta} \Big\} \end{aligned} \quad (13)$$

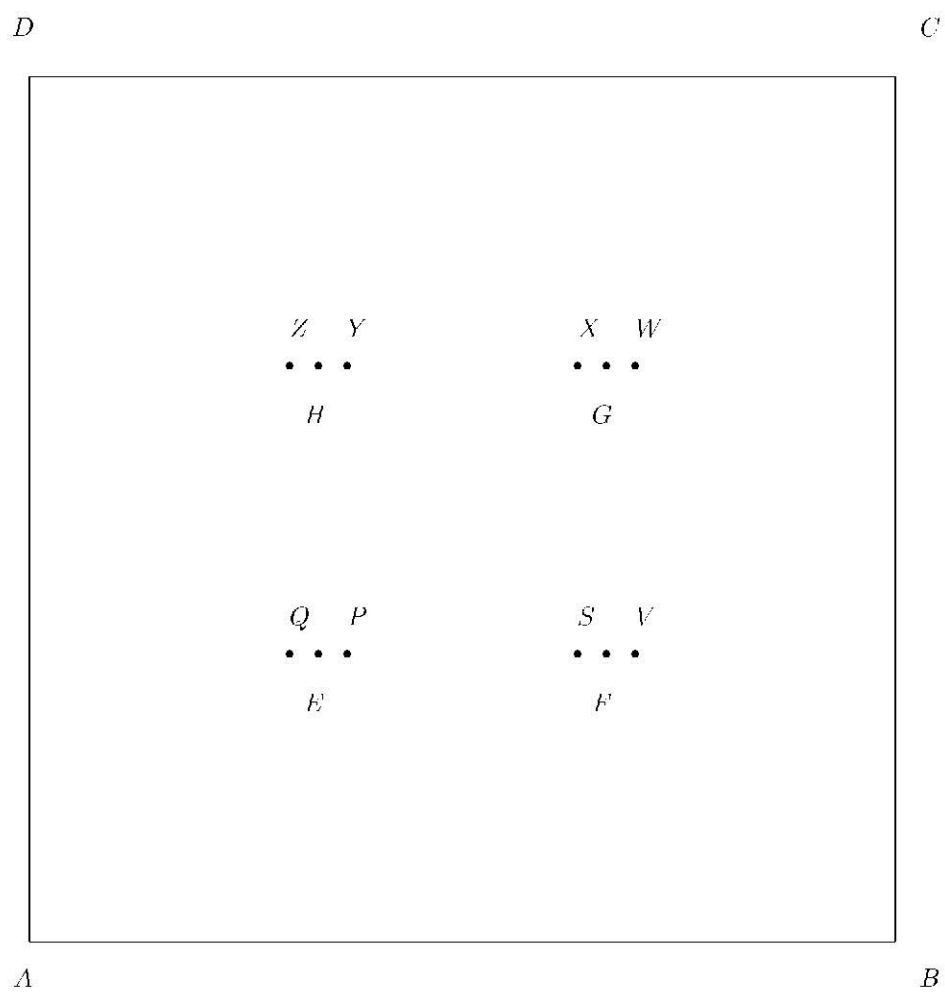


Figure 5: location of quadrature nodes

where

$$h_x = \frac{x_{l+1} - x_l}{3}, \quad h_y = \frac{y_{m+1} - y_m}{3} \quad (14)$$

and δ , the spacing for the centered differences was arbitrarily chosen as

$$\delta = .05(x_{l+1} - x_l). \quad (15)$$

Similarly, we can approximate the second integral in K_{ij} except that now the points will be $\delta = .05(y_{m+1} - y_m)$ units above and below the four points E, F, G, H .

4. Semi-Lagrangian Finite Elements

Semi-Lagrangian schemes belong to the general class of upwinding methods. For hyperbolic equations, upwinding methods incorporate characteristic information into the numerical method. In Lagrangian schemes, the evolution of the system is monitored by following specific fluid particles through space. As a result, Lagrangian schemes allow larger time steps than Eulerian. The problem with fully Lagrangian schemes is that an initially regularly spaced set of particles will generally evolve into irregularly spaced particles. As a result, some important features in the flow may not be captured properly. Semi-Lagrangian schemes combine the best of both worlds: the regular resolution of an Eulerian scheme and the high stability of a Lagrangian method. The idea is to choose a different set of particles such that at the end of the time step, they arrive at points on a regular Cartesian grid. The departure points of the particles are determined by an iterative process using the interpolated velocity vector from the previous time.

A semi-Lagrangian formulation of (1)

$$\frac{c^+ - c^-}{2\Delta t} + \frac{1}{2} \left[(cu_x + cv_y)^+ + (cu_x + cv_y)^- \right] = 0 \quad (16)$$

where c^+ is the solution at the grid points at time $t + \Delta t$, c^- is the solution at time $(t - \Delta t)$ at those points arriving at the grid points at time $t + \Delta t$. Since one requires two previous time levels, the program uses Matsuno's (see e.g. Haltiner and Williams [12]) method to get the first time step.

In the appendix, we bring plots of the solution for the cone test (see e.g. Zlatev [1]) using Eulerian finite elements with explicit, Crank-Nicholson and fully implicit time discretizations as well as the semi-Lagrangian method.

5. Stability Analysis

There are four rectangles having a vertex in common, as indicated in the next figure. The approximate solution at the vertices of the rectangles may be obtained by solving the following first order ordinary differential equation (see Neta and Williams [6] for the one dimensional advection case).

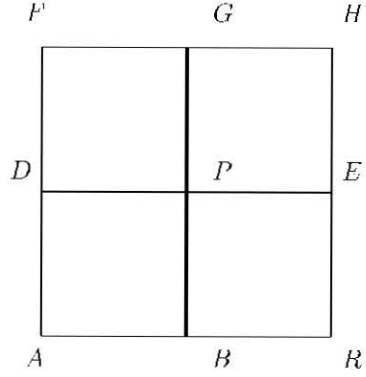


Figure 6: rectangular elements having a common vertex

$$\begin{aligned}
\dot{c}(P) &+ \frac{1}{4} [\dot{c}(G) + \dot{c}(B) + \dot{c}(D) + \dot{c}(E)] \\
&+ \frac{1}{16} [\dot{c}(F) + \dot{c}(H) + \dot{c}(A) + \dot{c}(R)] \\
&+ \frac{3}{16} u \frac{1}{\Delta x} \{c(H) - c(F) + c(R) - c(A) + 4[c(E) - c(D)]\} \\
&+ \frac{3}{16} v \frac{1}{\Delta y} \{c(H) - c(R) + c(F) - c(A) + 4[c(G) - c(B)]\} = 0
\end{aligned} \tag{17}$$

Substitute a Fourier mode

$$c(x, y, t) = A(t)e^{i(\mu x + \nu y)} \tag{18}$$

in (17) to get

$$\begin{aligned}
\dot{A}(t) &\left\{ 1 + \frac{1}{4}(2 \cos \mu \Delta x + 2 \cos \nu \Delta y) + \frac{1}{16} 4 \cos \mu \Delta x \cos \nu \Delta y \right\} \\
&+ \frac{3}{16} \frac{u}{\Delta x} A(t) \left\{ 4 + 2 \cos \nu \Delta y \right\} 2i \sin \mu \Delta x \\
&+ \frac{3}{16} \frac{v}{\Delta y} A(t) \left\{ 4 + 2 \cos \mu \Delta x \right\} 2i \sin \nu \Delta y = 0
\end{aligned} \tag{19}$$

or

$$\dot{A}(t) + 3i \left\{ \frac{u}{\Delta x} \frac{\sin \mu \Delta x}{2 + \cos \mu \Delta x} + \frac{v}{\Delta y} \frac{\sin \nu \Delta y}{2 + \cos \nu \Delta y} \right\} A(t) = 0. \quad (20)$$

For the special case of flow along the x or y axis one of the terms in braces will drop. For flow along the diagonal

$$\frac{u}{v} = \frac{\Delta y}{\Delta x} \quad (21)$$

we have

$$\dot{A}(t) + 3i \frac{u}{\Delta x} \left\{ \frac{\sin \mu \Delta x}{2 + \cos \mu \Delta x} + \frac{\sin \nu \Delta y}{2 + \cos \nu \Delta y} \right\} A(t) = 0. \quad (22)$$

In general, the ordinary differential equation becomes

$$\dot{A}(t) + i\sigma A(t) = 0, \quad (23)$$

where σ is 3 times the term in braces in (20). For the leap-frog time discretization

$$A_{n+1} - A_{n-1} + 2i\sigma \Delta t A_n = 0 \quad (24)$$

we have

$$\lambda_{1,2} = -i\sigma \Delta t \pm \sqrt{1 - \sigma^2 (\Delta t)^2}, \quad (25)$$

and thus for stability ($|\lambda| \leq 1$), we must have

$$|\sigma \Delta t| \leq 1. \quad (26)$$

If we let

$$\begin{aligned} U &= \max(|u|, |v|) \\ \delta &= \min(\Delta x, \Delta y) \end{aligned} \quad (27)$$

then the method is stable if

$$\frac{\Delta t}{\delta} \leq \frac{2}{3U}. \quad (28)$$

This is the CFL condition.

6. Fourier Transform

The Fourier transform of a function $c(x, y, t)$ is given by

$$\mathcal{F}\{c\} = \hat{c}(k, l, t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} c(x, y, t) e^{-i(kx + ly)} dx dy \quad (29)$$

Taking the Fourier transform of the linearization of (1), one gets the initial value problem

$$\frac{d\hat{c}}{dt} + i(ku + lv)\hat{c} = 0, \quad (30)$$

$$\hat{c}(k, l, 0) = \hat{c}_0(k, l). \quad (31)$$

The solution of which is given by

$$\hat{c}(k, l, t) = \hat{c}_0(k, l)e^{i\nu t}, \quad (32)$$

where

$$\nu = -(ku + lv). \quad (33)$$

To get the solution $c(x, y, t)$ in the physical domain, we have to take the inverse Fourier transform and use the convolution theorem. This yields the well known solution

$$c(x, y, t) = c_0(x - ut, y - vt). \quad (34)$$

In order to obtain the Fourier transform of the approximate solution, recall that

$$\int_{-\infty}^{\infty} u(x + \Delta x, y, t) e^{-ikx} dx = e^{ik\Delta x} \hat{u}(k, y, t). \quad (35)$$

Applying Fourier transform to (17), one gets

$$\frac{\hat{c}}{dt} + \left[iu \frac{3}{\Delta x} \frac{\sin k\Delta x}{2 + \cos k\Delta x} + iv \frac{3}{\Delta y} \frac{\sin l\Delta y}{2 + \cos l\Delta y} \right] \hat{c} = 0. \quad (36)$$

Compare (36) and (30), to find that k, l are replaced by σ_x, σ_y respectively, where

$$\sigma_x = \frac{3}{\Delta x} \frac{\sin k\Delta x}{2 + \cos k\Delta x}, \quad \sigma_y = \frac{3}{\Delta y} \frac{\sin l\Delta y}{2 + \cos l\Delta y}.$$

Note that as $\Delta x \rightarrow 0$, $\sigma_x \rightarrow k$ and as $\Delta y \rightarrow 0$, $\sigma_y \rightarrow l$, thus at the limit (36) becomes (30). In fact

$$\sigma_x \sim k - \frac{1}{180} k^5 \Delta x^4 - \frac{1}{1512} k^7 \Delta x^6 + O(k^9)$$

$$\sigma_y \sim l - \frac{1}{180} l^5 \Delta y^4 - \frac{1}{1512} l^7 \Delta y^6 + O(l^9)$$

The solution of (36) with the same initial value is given by

$$\hat{c}(k, l, t) = \hat{c}_0(k, l) e^{i\hat{\nu}t}, \quad (37)$$

where

$$\hat{\nu} = -(\sigma_x u + \sigma_y v). \quad (38)$$

The inverse transform is given by

$$c(x, y, t) = \frac{1}{4\pi^2} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \hat{c}_0(k, l) e^{-i(u\sigma_x + v\sigma_y)t} e^{i(kx + ly)} dk dl \quad (39)$$

or by using convolution

$$c(x, y, t) = c_0 * \mathcal{F}^{-1} \left\{ e^{-i(u\sigma_x + v\sigma_y)t} \right\}. \quad (40)$$

7. Program Notes

The program can run semi-Lagrangian as well as Eulerian finite elements. In the Eulerian case the time differencing is one of the following:

- Explicit
- Crank Nicholson
- fully implicit

where the resulting linear system of equations is solved by the conjugate gradient method with symmetric SOR preconditioning (see e.g. Ortega [15]).

The four lines of input contain:

1. I, J
2. x_L, x_R, y_L, y_U
3. $\Delta t, T$ (final time of integration), IPLOT (number of time steps between solution plots).
4. θ (a parameter dictating the time integrator). This is needed only for Eulerian finite elements.

Here we include the input file and the programs used to test the Eulerian and semi-Lagrangian finite element methods. First we give the input file. The first line contains the number of grid points in the x and y directions. The second line describe the rectangular domain on which the problem is solved. The interval for x is given followed by the interval for y . The third line gives the time step Δt , the final time of integration, and the number of time steps between plots. For Eulerian finite elements, we have a fourth line with the value of θ .

Here is the input file for the explicit time discretization.

```
21 21
0.0 2.0 0.0 2.0
0.0125 1.0 10
0.
```

Here is the program for the Eulerian finite elements.

```

*-----*
*This program solves the 2D Advection Equation
*       $dc/dt + d/dx(cu) + d/dy(cv) = 0$ 
*on a square domain using Periodic B.C's
*in both x and y using Bilinear Rectangular Finite Elements
*and THETA Time-Integration Algorithms.
* THETA=0  -> FORWARD EULER  (Explicit)
* THETA=1/2 -> CRANK-NICOLSON (Semi-Implicit)
* THETA=2/3 -> GALERKIN      (Semi-Implicit)
* THETA=1   -> BACKWARD EULER (Implicit)
*Written by F.X. Giraldo on 3/95
*      NRC Fellow
*      Department of Mathematics
*      Naval Postgraduate School
*      Monterey, CA 93940
*-----*

      program fem_advect
      implicit real*8(a-h,o-z)
      parameter ( imax=21 )

c
c mxpoi  max number of points
c mxele  max number of elements
c mxbou  max number of boundary points
c nd      max number of vertices for each elements
c
      parameter ( mx=imax*imax, mxpoi=mx, mxele=mx, mxbou=mx/5, nd=4 )
c
c      global matrices
c
      dimension alhs(mxpoi,mxpoi), arhs(mxpoi,mxpoi), b(mxpoi)
      dimension coord(mxpoi,2)
      integer intma(mxele,nd), iboun(mxbou,4), node(imax,imax)
c
c      u velocity arrays
c
      dimension u(mxpoi)
c
c      v velocity arrays
c
      dimension v(mxpoi)
c
c      phi arrays

```

```

c      dimension phip(mxpoi), phi0(mxpoi)
c
c      Read the Input Variables and create the Grid
c
c
c      input file contains 4 lines
c on first: number of grid points in x (nx) and y (ny) direction
c on second: range of x values (xmin, xmax),
c           range of y values (ymin,ymax)
c on third: delta t,
c           final time of integration (time_final),
c           number of times steps between plots (iplot)
c on fourth: theta (see above)
c
c      call init(phi0,u,v,node,coord,intma,iboun,npoin,nelem,
$           nboun,xmin,xmax,ymin,ymax,ym,nx,ny,nd,dx,dy,dt,
$           ntime,theta,mxpoi,mxele,mxbou,imax,iplot)
c      pi=4.0*atan(1.0)
c      open(1,file='matlab.out')
c
c isets = number of time steps at which solution is plotted
c
c      isets=ntime/iplot + 2
c      write(1,*)isets
c
c      always plot initial condition
c
c      call output(phi0,u,v,npoin,time,nx,ny,mxpoi)
c
c      begin the time marching
c
c      time=0.0
c
c      CREATE THE STIFFNESS MATRIX ONCE
c
c      call lhs(alhs,arhs,coord,intma,iboun,node,u,v,npoin,nelem,
$           nboun,nx,ny,nd,dx,dy,dt,theta,mxpoi,mxele,mxbou,imax)
c      do i=1,npoin
c          do j=1,npoin
c              write(2,*)i,j,alhs(i,j)
c          end do
c      end do
c
c

```



```

c      TIME MARCH
c
      do itime=1,ntime

          time=time + dt
          write(*, '(' timestep time = ',i5,2x,e12.4)') itime,time/(2.0*pi)

c
c      Solve for the GeoPotential
c
      call rhs(b,arhs,phi0,coord,intma,iboun,node,
$          npoin,nelem,nboun,nx,ny,nd,dx,dy,
$          mxpoi,mxele,mxbou,imax)
      if (theta.eq.0.0) then
          call solve_explicit(alhs,phip,b,npoin,mxpoi)
      else if (theta.ne.0.0) then
          call pcgm(alhs,phip,b,npoin,mxpoi)
      endif

c
c      Enforce B.C.s Explicitly
c
      do i=1,nx
          j1=node(i,1)
          jny=node(i,ny)
          phip(j1)=phip(jny)
      end do

      do j=1,ny
          i1=node(1,j)
          inx=node(nx,j)
          phip(i1)=phip(inx)
      end do

c
c      Update
c
      do ip=1,npoin
          phi0(ip)=phip(ip)
      end do

c
c      check printing status
c
      if (mod(itime,iplot).eq.0)
$          call output(phi0,u,v,npoin,time,nx,ny,mxpoi)

      end do

```

```

1000 continue
      call output(phi0,u,v,npoin,time,nx,ny,mxpoi)
      close(1)
      stop
      end

*-----*
*This subroutine writes the output.  It is currently set only to
*print the concentration (or color) function at each node point.
*Written by F.X. Giraldo on 2/95
*-----*

      subroutine output(phi,u,v,npoin,time,nx,ny,mxpoi)
      implicit real*8(a-h,o-z)
      dimension phi(mxpoi), u(mxpoi), v(mxpoi)

      pi=4.0*atan(1.0)
      write(1,'(2(i6,1x),e16.8)') ,nx,ny,time/(2.0*pi)
      write(1,'(e12.4)')(phi(ip), ip=1,npoin)
      return
      end

*-----*
*This subroutine reads in the input file.
*The info read is the number of grid points (in x and y), the domain,
*the time step, the final time, and the number of time steps for plotting.
*Written by F.X. Giraldo on 2/95
*-----*

      subroutine init(phi0,u0,v0,node,coord,intma,iboun,npoin,nelem,
$              nboun,xmin,xmax,ymin,ymax,ym,nx,ny,nd,dx,dy,dt,
$              ntime,theta,mxpoi,mxele,mxbou,imax,iplot)
      implicit real*8(a-h,o-z)
      dimension coord(mxpoi,2)
      dimension phi0(mxpoi), u0(mxpoi), v0(mxpoi)
      integer intma(mxele,nd), iboun(mxbou,4), node(imax,imax)

c
c   Read Input File
c
      read(*,*)nx,ny
      read(*,*)xmin,xmax,ymin,ymax
      read(*,*)dt,time_final,iplot
      read(*,*)theta

c
c   check bounds
c
      if (nx*ny.gt.mxpoi) then

```

```

        write(*, '(" Error! - Need to Enlarge MXPOI")')
        stop
    else if ((nx-1)*(ny-1).gt.mxele) then
        write(*, '(" Error! - Need to Enlarge MXELE")')
        stop
    else if (2*(nx-1)+2*(ny-1).gt.mxbou) then
        write(*, '(" Error! - Need to Enlarge MXBOU")')
        stop
    endif
endif

c
c  set some constants
c
pi=4.0*atan(1.0)
ntime=nint(time_final/dt)
dt=dt*2.0*pi
time_final=time_final*2.0*pi
xm=0.5*(xmax+xmin)
ym=0.5*(ymax+ymin)
dx=(xmax-xmin)/(nx-1)
dy=(ymax-ymin)/(ny-1)
xl=xmax-xmin
yl=ymax-ymin
w=1.0
cx=0.25*xl
cy=0.50*yl
h=100.0
rc=0.125*xl
velmax=-1e5

c
c  set the Initial Conditions
c
ip=0
do j=1,ny
    y=ymin + real(j-1)*dy
    do i=1,nx
        x=xmin + real(i-1)*dx
        ip=ip+1
        r=sqrt( (x-cx)**2 + (y-cy)**2 )
        phi0(ip)=0.0
        if (r.lt.rc) then
            phi0(ip)=h*(1.0-r/rc)
        endif
        u0(ip)=+(y-ym)
        v0(ip)=- (x-xm)
    enddo
enddo

```

```

        vel1=u0(ip)**2 + v0(ip)**2
        velmax=max(velmax,vel1)
    end do
end do
cfl=dt*sqrt(velmax/(dx**2 + dy**2))
print*, ' ** CFL = ',cfl
c
c  GENERATE COORD
c
npoin=nx*ny
ip=0
do j=1,ny
    do i=1,nx
        ip=ip+1
        node(i,j)=ip
        coord(ip,1)=xi + dx*real(i-1)
        coord(ip,2)=yi + dy*real(j-1)
    end do
end do
c
c  GENERATE INTMA
c
nelem=(nx-1)*(ny-1)
ie=0
do j=1,ny-1
    do i=1,nx-1
        ie=ie+1
        intma(ie,1)=node(i,j)
        intma(ie,2)=node(i+1,j)
        intma(ie,3)=node(i+1,j+1)
        intma(ie,4)=node(i,j+1)
    end do
end do
c
c  GENERATE IBOUN
c
nboun=2*(nx-1) + 2*(ny-1)
ib=0
c
bottom (y=ymin)
ie=1
do i=1,nx-1
    ib=ib+1
    iboun(ib,1)=node(i,1)

```

```

        iboun(ib,2)=node(i+1,1)
        iboun(ib,3)=ie
        iboun(ib,4)=1
        ie=ie+1
    end do

c    top (y=ymax)
    ie=nelem
    do i=nx,2,-1
        ib=ib+1
        iboun(ib,1)=node(i,ny)
        iboun(ib,2)=node(i-1,ny)
        iboun(ib,3)=ie
        iboun(ib,4)=1
        ie=ie-1
    end do

c    right (x=xmax)
    ie=(nx-1)
    do j=1,ny-1
        ib=ib+1
        iboun(ib,1)=node(nx,j)
        iboun(ib,2)=node(nx,j+1)
        iboun(ib,3)=ie
        iboun(ib,4)=2
        ie=ie + (nx-1)
    end do

c    left (x=xmin)
    ie=nelem - (nx-1) + 1
    do j=ny,2,-1
        ib=ib+1
        iboun(ib,1)=node(1,j)
        iboun(ib,2)=node(1,j-1)
        iboun(ib,3)=ie
        iboun(ib,4)=2
        ie=ie - (nx-1)
    end do

    return
end

*-----*
*This subroutine constructs the LHS matrix for Bilinear Rectangular
*Elements for the Advection Equation with Periodic

```

*East-West and North-South Boundary Conditions.

*Written by F.X. Giraldo on 2/95

```

*-----*
      subroutine lhs(alhs,arhs,coord,intma,iboun,node,u,v,npoin,nelem,
$          nboun,nx,ny,nd,dx,dy,dt,theta,mxpoi,mxele,mxbou,imax)
      implicit real*8(a-h,o-z)
      parameter (mx=3000)

c
c      global arrays
c
      dimension alhs(mxpoi,mxpoi), arhs(mxpoi,mxpoi)
      dimension coord(mxpoi,2), u(mxpoi), v(mxpoi)
      integer intma(mxele,nd), node(imax,imax), iboun(mxbou,4)

c
c      local coordinate system for a CCW ordered rectangle
c
      dimension xi(4), eta(4), a_temp(mx)
      data xi /-1, 1, 1,-1 /
      data eta /-1,-1, 1, 1 /

c
c      initialize the global matrix
c
      do j=1,npoin
do i=1,npoin
      alhs(i,j)=0.0
      arhs(i,j)=0.0
      end do
      end do

c
c      loop thru the elements
c
      do ie=1,nelem

c
c      assemble element matrix      cm == consistent mass matrix
c                                  ckc == conduction-like matrix

      do i=1,nd
          ii=intma(ie,i)
          cm_lump=0.0
          do j=1,nd
              jj=intma(ie,j)
              cm=(2.0+2.0/3.0*xi(i)*xi(j))*(2.0+2.0/3.0*eta(i)*eta(j))
              ckc=0.0
              cky=0.0
              do k=1,nd

```

```

        uk=u(intma(ie,k))
        vk=v(intma(ie,k))
        ckx=ckx + uk*
$ (2.0*xi(i) + 2.0/3.0*xi(i)*xi(j)*xi(k))*
$ (2.0+2.0/3.0*(eta(i)*eta(j) + eta(i)*eta(k) + eta(j)*eta(k)))
        cky=cky + vk*
$ (2.0*eta(i) + 2.0/3.0*eta(i)*eta(j)*eta(k))*
$ (2.0+2.0/3.0*(xi(i)*xi(j) + xi(i)*xi(k) + xi(j)*xi(k)))
        end do
        cmm=dx*dy/64.0*cm
        ck=dy/128.0*ckx + dx/128.0*cky
        alhs(ii,jj)=alhs(ii,jj) + 1.0/dt*cmm - theta*ck
        arhs(ii,jj)=arhs(ii,jj) + 1.0/dt*cmm + (1.0-theta)*ck
    end do
end do
c
c Account for Periodicity
c
do i=1,nx
    j1=node(i,1)
    jny=node(i,ny)
    do jj=1,npoin
        a_temp(jj)=alhs(j1,jj) + alhs(jny,jj)
    end do
    do jj=1,npoin
        alhs(j1,jj)=a_temp(jj)
        alhs(jny,jj)=a_temp(jj)
    end do
    alhs(j1,j1)=a_temp(j1) + a_temp(jny)
    alhs(jny,jny)=a_temp(j1) + a_temp(jny)
end do

do j=1,ny
    i1=node(1,j)
    inx=node(nx,j)
    do jj=1,npoin
        a_temp(jj)=alhs(i1,jj) + alhs(inx,jj)
    end do
    do jj=1,npoin
        alhs(i1,jj)=a_temp(jj)
        alhs(inx,jj)=a_temp(jj)
    end do
    alhs(i1,i1)=a_temp(i1) + a_temp(inx)

```

```

        alhs(inx,inx)=a_temp(il) + a_temp(inx)
    end do

c
c    If theta=0, then Lump.
c
    if (theta.eq.0.0) then
        do i=1,npoin
            asum=0.0
            do j=1,npoin
                asum=asum + alhs(i,j)
                alhs(i,j)=0.0
            end do
            alhs(i,i)=asum
        end do
    endif

    return
end

*-----*
*The next 4 Subroutines solve a linear system  $[A]\{x\}=\{b\}$ , for n unknowns using
*the CONJUGATE GRADIENT METHOD with an SSOR preconditioner.
*The variable w determines the preconditioner. for w=1 it is
*Symmetric Jacobi and for w>1 it is Symmetric SOR (SSOR).
*Written by F.X. Giraldo on 4/14/90
*Modified for any type of spd Matrix A on 2/95
*-----*

subroutine pcgm(a,x,b,n,mx)
    implicit real*8(a-h,o-z)
    parameter (nmax=3000, kmax=15, w=1.0)
    dimension a(mx,mx), b(mx), x(mx)
    dimension r(nmax), rw(nmax), p(nmax), ap(nmax)

    do i=1,n
        sum=0.0
        do j=1,n
            sum=sum + a(i,j)*x(j)
        end do
        r(i)=b(i) - sum
    end do

    call ssor(r,rw,a,n,w,nmax,mx)

    do i=1,n
        p(i)=rw(i)
    end do

```



```

end do

call ip(rop,rw,r,n,nmax)

do k=1,kmax

    do i=1,n
        sum=0.0
        do j=1,n
            sum=sum + a(i,j)*p(j)
        end do
        ap(i)=sum
    end do

    alfden=0.0
    do i=1,n
        alfden=alfden + p(i)*ap(i)
    end do
    alf=-rop/alfden

    do i=1,n
        x(i)=x(i)-alf*p(i)
    end do

    do i=1,n
        r(i)=r(i) + alf*ap(i)
    end do

    call convtest(rop,r,n,flag,nmax)
    if (flag .eq. 1.0) goto 200

    call ssor(r,rw,a,n,w,nmax,mx)
    call ip(rnp,rw,r,n,nmax)

    beta=rnp/rop
    do i=1,n
        p(i)=rw(i) + beta*p(i)
    end do
    rop=rnp
end do
k=k-1
print*, '      No Convergence'
200 continue

```

```

        print*, '      SSOR PCG Loops = ', k
        return
    end

*-----*
    subroutine ssor(r,rw,a,n,w,nmax,mx)
    implicit real*8(a-h,o-z)
    dimension r(nmax), rw(nmax)
    dimension a(mx,mx)

    * symmetric sor on the residual
    * zeroing the wiggle residual

        do i=1,n
            rw(i)=0.0
        end do

    * up sweep

        do i=1,n
            sum=0.0
            do j=1,n
                sum=sum + a(i,j)*rw(j)
            end do
            rw(i)=rw(i) + w/a(i,i)*( r(i) - sum )
        end do

    * down sweep

        do i=n,1,-1
            sum=0.0
            do j=1,n
                sum=sum + a(i,j)*rw(j)
            end do
            rw(i)=rw(i) + w/a(i,i)*( r(i) - sum )
        end do
        return
    end

*-----*
    subroutine ip(f,fw,fr,n,nmax)
    implicit real*8(a-h,o-z)
    dimension fw(nmax), fr(nmax)
    f=0.0
    do i=1,n
        f=f + fw(i)*fr(i)
    end do

```

```

end do
return
end
*-----*
subroutine convtest(rop,r,n,flag,nmax)
implicit real*8(a-h,o-z)
dimension r(nmax)

emin=1.0e-6
flag=0.0
rtest=0.0
if (rop .lt. emin) then
do i=1,n
rtest=rtest + r(i)**2.0
end do
if (rtest .lt. emin) flag=1.0
endif
return
end
*-----*
*This subroutine Builds the RHS vector for Bilinear Rectangular Finite
*Elements for the 2D SLSI Shallow Water Equations with Periodic
*West-East and North-South Boundaries.
*Written by F.X. Giraldo on 2/95
*-----*
subroutine rhs(b,arhs,phi0,coord,intma,iboun,node,
$ npoin,nelem,nboun,nx,ny,nd,dx,dy,
$ mxpoi,mxele,mxbou,imax)
implicit real*8(a-h,o-z)
parameter (mx=3000)

c
c global arrays
c
dimension b(mxpoi), arhs(mxpoi,mxpoi), phi0(mxpoi)
dimension coord(mxpoi,2)
integer intma(mxele,nd), iboun(mxbou,4), node(imax,imax)

c
c local coordinates for a CCW ordered Rectangle
c
dimension xi(4), eta(4)
data xi /-1, 1, 1,-1/
data eta /-1,-1, 1, 1/

c
c initialize the right hand side

```

```

c
      do ip=1,npoin
b(ip)=0.0
      end do
c
c   Construct the RHS Vector
c
      do i=1,npoin
        do j=1,npoin
          b(i)=b(i) + arhs(i,j)*phi0(j)
        end do
      end do
c
c   Account for Periodicity
c
      do i=1,nx
        j1=node(i,1)
        jny=node(i,ny)
        b_temp=b(j1) + b(jny)
        b(j1)=b_temp
        b(jny)=b_temp
      end do
      do j=1,ny
        i1=node(1,j)
        inx=node(nx,j)
        b_temp=b(i1) + b(inx)
        b(i1)=b_temp
        b(inx)=b_temp
      end do
      return
      end
*-----*
*This subroutine solves a Linear NxN system where the Coefficient
*Matrix is diagonal.
*Written by F.X. Giraldo on 4/14/90
*-----*
      subroutine solve_explicit(a,x,b,n,mx)
      implicit real*8(a-h,o-z)
      dimension a(mx,mx), b(mx), x(mx)
      do i=1,n
        x(i)=b(i)/a(i,i)
      end do
      return
      end

```

Here is the program for the semi-Lagrangian finite elements.

```

*-----*
*This program solves the 2D Advection Equation
*       $dc/dt + d/dx(cu) + d/dy(cv) = 0$ 
*on a square domain using Periodic B.C.'s
*in both x and y and using Semi-Implicit Semi-Lagrangian
*Bilinear Rectangular Finite Elements.
*Written by F.X. Giraldo on 3/95
*      NRC Fellow
*      Department of Mathematics
*      Naval Postgraduate School
*      Monterey, CA 93940
*-----*

      program slt_advect
      implicit real*8(a-h,o-z)
      parameter ( imax=21 )

c
c mxpoi  max number of points
c mxele  max number of elements
c mxbou  max number of boundary points
c nd     max number of vertices for each elements
c
      parameter ( mx=imax*imax, mxpoi=mx, mxele=mx, mxbou=mx/5, nd=4 )
c
c      global matrices
c
      dimension a(mxpoi,mxpoi), b(mxpoi)
      dimension f(mxpoi)
      dimension coord(mxpoi,2)
      dimension cmat(mxpoi)
      integer intma(mxele,nd), iboun(mxbou,4)

c
c      u velocity arrays
c
      dimension um(mxpoi),      u0(mxpoi),      up(mxpoi)
      dimension                  u0_x(mxpoi)

c
c      v velocity arrays
c
      dimension vm(mxpoi),      v0(mxpoi),      vp(mxpoi)
      dimension                  v0_y(mxpoi)

c
c      phi arrays

```

```

c      dimension phim(mxpoi),    phi0(mxpoi),    phip(mxpoi)
c
c      departure point arrays
c
c      dimension alfm(mxpoi,2),  alf0(mxpoi,2)
c
c      spline derivative arrays
c
c      dimension dphim(mxpoi)
c      dimension du0(mxpoi), du0_x(mxpoi)
c      dimension dv0(mxpoi), dv0_y(mxpoi)
c
c      Auxiliary Matrices
c
c      integer node(imax,imax)
c
c      Read the Input Variables and create the Grid
c
c      input file contains 4 lines
c on first: number of grid points in x (nx) and y (ny) direction
c on second: range of x values (xmin, xmax),
c           range of y values (ymin, ymax)
c on third: delta t,
c           final time of integration (time_final),
c           number of times steps between plots (iplot)
c
c      call init(phi0,u0,v0,node,coord,alf0,intma,iboun,npoin,
$           nelem,nboun,xmin,xmax,ymin,ymax,nx,ny,dx,dy,dt,
$           ntime,ym,nd,mxpoi,mxele,mxbou,imax,iplot)
c
c      Construct the Lumped Mass Matrix used for the derivative computation
c
c      call get_geom(nelem,npoin,intma,coord,cmat,node,
$           nx,ny,nd,dx,dy,mxpoi,mxele,imax)
c      time=0.0
c      pi=4.0*atan(1.0)
c      open(1,file='matlab.out')
c
c isets = number of time steps at which solution is plotted
c
c      isets=ntime/iplot + 2
c      write(1,*)isets
c

```

```

c  always plot initial condition
c
c      call output(phi0,u0,v0,npoin,time,nx,ny,mxpoi)
c
c  begin the time marching
c
c  Do the 1st Time-step Integration to get 2-time levels
c
c      do itime=1,1
c          time=time + dt
c          dtime=time/(2.0*pi)
c          write(*, '(' timestep time = ',i5,2x,e12.4)')itime,dtime
c          call matsuno(phim,phi0,phip,um,u0,up,
$              vm,v0,vp,coord,intma,npoin,nelem,dt,dx,dy,
$              node,nx,ny,ym,nd,mxpoi,mxele,imax)
c          call update(phim,phi0,phip,um,u0,up,vm,v0,vp,alfm,alf0,
$              npoin,mxpoi)
c      end do
c
c  TIME MARCH
c
c      do itime=2,ntime
c
c          time=time + dt
c          dtime=time/(2.0*pi)
c          write(*, '(' timestep time = ',i5,2x,e12.4)')itime,dtime
c
c      *
c      *      1st, DETERMINE DEPARTURE POINT
c      *
c          call splie2(du0,u0,coord,nx,ny,node,mxpoi,imax)
c          call splie2(dv0,v0,coord,nx,ny,node,mxpoi,imax)
c          call depart(alf0,alfm,coord,u0,du0,v0,dv0,
$              npoin,dt,xmin,xmax,ymin,ymax,
$              nd,mxpoi,node,nx,ny,imax)
c
c      *
c      *      2nd, COMPUTE DERIVATIVES
c      *
c          call deriv_x(u0_x,u0,intma,cmat,node,npoin,nelem,
$              nx,ny,nd,dy,mxpoi,mxele,imax)
c          call deriv_y(v0_y,v0,intma,cmat,node,npoin,nelem,
$              nx,ny,nd,dx,mxpoi,mxele,imax)
c
c      *
c      *      3rd, INTERPOLATE PHIM, UM_X=UO_X, VM_Y=VO_Y
c      *      at the departure point = X - 2*ALPHA

```

```

*
      call splie2(dphim,phim,coord,nx,ny,node,mxpoi,imax)
      call splie2(du0_x,u0_x,coord,nx,ny,node,mxpoi,imax)
      call splie2(dv0_y,v0_y,coord,nx,ny,node,mxpoi,imax)
c
c      INTERPOLATE THE RIGHT HAND SIDE FUNCTION
c
      call interp(f,phim,u0_x,v0_y,dphim,du0_x,dv0_y,coord,
$              alf0,node,npoin,dt,xmin,xmax,ymin,ymax,ym,
$              nx,ny,nd,mxpoi,imax)
      do ip=1,npoin
          phip(ip)=f(ip)
          up(ip)=u0(ip)
          vp(ip)=v0(ip)
      end do
c
c      Update the values
c
      call update(phim,phi0,phip,um,u0,up,vm,v0,vp,alfm,alf0,
$              npoin,mxpoi)
c
c      check time for printing output
c
      if (mod(itime,iplot).eq.0)
$          call output(phi0,u0,v0,npoin,time,nx,ny,mxpoi)

      end do

1000 continue

      call output(phi0,u0,v0,npoin,time,nx,ny,mxpoi)
      close(1)
      stop
      end
*-----*
*This subroutine finds the departure point
* ALPHA1=DT*U(X-ALPHA1,Y-ALPHA2,T) ALPHA2=DT*V(X-ALPHA1,Y-ALPHA2,T)
*Written by F.X. Giraldo on 2/95
*-----*
      subroutine depart(alf0,alfm,coord,u0,du0,v0,dv0,
$              npoin,dt,xmin,xmax,ymin,ymax,
$              nd,mxpoi,node,nx,ny,imax)
      implicit real*8(a-h,o-z)
      dimension alf0(mxpoi,2), alfm(mxpoi,2)

```



```

dimension u0(mxpoi), du0(mxpoi)
dimension v0(mxpoi), dv0(mxpoi)
dimension coord(mxpoi,2)
integer node(imax,imax)

do ip=1,npoin
  alpha1=alfm(ip,1)
  alpha2=alfm(ip,2)
  do k=1,3
    xd=coord(ip,1) - alpha1
    yd=coord(ip,2) - alpha2
    if (xd.lt.xmin) xd=xmax - (xmin - xd)
    if (xd.gt.xmax) xd=xmin + (xd - xmax)
    if (yd.lt.ymin) yd=ymax - (ymin - yd)
    if (yd.gt.ymax) yd=ymin + (yd - ymax)

    if ( (xd.lt.xmin.or.xd.gt.xmax) .or.
$      (yd.lt.ymin.or.yd.gt.ymax) ) then
      print*, ' Error in DEPART'
      print*, 'XD out of Range = ',xd,yd
      print*, 'ip alpha = ',ip,alpha1,alpha2
      stop
    endif

    call splin2(u,coord,u0,du0,xd,yd,nx,ny,node,mxpoi,imax)
    call splin2(v,coord,v0,dv0,xd,yd,nx,ny,node,mxpoi,imax)

    alpha1=dt*u
    alpha2=dt*v
  end do
  alf0(ip,1)=alpha1
  alf0(ip,2)=alpha2
end do

return
end

*-----*
*This subroutine computes the 4th Order Accurate derivative WRT X of the
*variable UNKNO and stores it in DERIP using Bilinear Rectangular
*Finite Elements
*Written by F.X. Giraldo on 2/95
*-----*

subroutine deriv_x(derip,unkno,intma,cmat,node,npoin,nelem,
$ nx,ny,nd,dy,mxpoi,mxele,imax)

```

```

        implicit real*8(a-h,o-z)
c
c      global arrays
c
        dimension derip(mxpoi), unkno(mxpoi)
        dimension cmat(mxpoi)
        integer intma(mxele,nd), node(imax,imax)
c
c      local coordinate system for a CCW ordered rectangle
c
        dimension xi(4), eta(4)
        data xi / -1, 1, 1,-1 /
        data eta / -1,-1, 1, 1 /
c
c      initialize arrays
c
        do ip=1,npoin
            derip(ip)=0.0
        end do
c
c      Loop thru the Elements and find the 1st DERIVATIVES
c
        do ie=1,nelem
            do i=1,nd
                phi_x=0.0
                ip=intma(ie,i)
                do k=1,nd
                    phik=unkno(intma(ie,k))
                    phi_x=phi_x + xi(k)*phik*( 2.0 + 2.0/3.0*eta(k)*eta(i) )
                end do
                derip(ip)=derip(ip) + dy/16.0*phi_x
            end do
        end do
c
c      Account for Periodicity
c
        do j=1,ny
            i1=node(1,j)
            inx=node(nx,j)
            derip_temp=derip(i1) + derip(inx)
            derip(i1)=derip_temp
            derip(inx)=derip_temp
        end do
c

```

```

c      Now multiply by the Inverse of the Lumped mass matrix
c
      do ip=1,npoin
        derip(ip)=derip(ip)*cmat(ip)
      end do

      return
      end

*-----*
*This subroutine computes the 4th Order Accurate derivative WRT Y of the
*variable UNKNO and stores it in DERIP using Bilinear Rectangular
*Finite Elements
*Written by F.X. Giraldo on 2/95
*-----*
      subroutine deriv_y(derip,unkno,intma,cmat,node,npoin,nelem,
$                        nx,ny,nd,dx,mxpoi,mxele,imax)
      implicit real*8(a-h,o-z)

c
c      global arrays
c
      dimension derip(mxpoi), unkno(mxpoi)
      dimension cmat(mxpoi)
      integer intma(mxele,nd), node(imax,imax)

c
c      local coordinate system for a CCW ordered rectangle
c
      dimension xi(4), eta(4)
      data xi / -1, 1, 1,-1 /
      data eta / -1,-1, 1, 1 /

c
c      initialize arrays
c
      do ip=1,npoin
        derip(ip)=0.0
      end do

c
c      Loop thru the Elements and find the 1st DERIVATIVES
c
      do ie=1,nelem
        do i=1,nd
          phi_y=0.0
          ip=intma(ie,i)
          do k=1,nd
            phik=unkno(intma(ie,k))

```

```

        phi_y=phi_y + eta(k)*phik*( 2.0 + 2.0/3.0*xi(k)*xi(i) )
    end do
    derip(ip)=derip(ip) + dx/16.0*phi_y
end do
c
c   Account for Periodicity
c
do j=1,ny
    i1=node(1,j)
    inx=node(nx,j)
    derip_temp=derip(i1) + derip(inx)
    derip(i1)=derip_temp
    derip(inx)=derip_temp
end do
c
c   Now multiply by the Inverse of the Lumped mass matrix
c
do ip=1,npoin
    derip(ip)=derip(ip)*cmat(ip)
end do

return
end

*-----*
*This subroutine computes the Inverse Lumped Mass Matrix
*for Bilinear Rectangular Finite Elements used for obtaining the
*4th Order Accurate 1st and 2nd derivatives in both X and Y.
*Written by F.X. Giraldo on 2/95
*-----*
subroutine get_geom(nelem,npoin,intma,coord,cmat,node,
$                nx,ny,nd,dx,dy,mxpoi,mxele,imax)
implicit real*8(a-h,o-z)
c
c   global arrays
c
dimension coord(mxpoi,2), cmat(mxpoi)
integer intma(mxele,nd), node(imax,imax)
c
c   Compute the inverse lumped mass matrix
c
do ip=1,npoin
    cmat(ip)=0.0
end do

```

```

do ie=1,nelem
  do in=1,nd
    ip=intma(ie,in)
    cmat(ip)=cmat(ip) + dx*dy/4.0
  end do
end do

c
c   Account for Periodicity in X
c
do j=1,ny
  i1=node(1,j)
  inx=node(nx,j)
  cmat_temp=cmat(i1) + cmat(inx)
  cmat(i1)=cmat_temp
  cmat(inx)=cmat_temp
end do

c
c   Invert the lumped mass matrix
c
do ip=1,npoin
  cmat(ip)=1.0/cmat(ip)
end do

return
end

*-----*
*This subroutine reads in the input file.
*The info read is the number of grid points (in x and y), the domain,
*the time step, the final time, and the number of time steps for plotting.
*Written by F.X. Giraldo on 2/95
*-----*

subroutine init(phi0,u0,v0,node,coord,alf0,intma,iboun,npoin,
$              nelem,nboun,xmin,xmax,ymin,ymax,nx,ny,dx,dy,dt,
$              ntime,ym,nd,mxpoi,mxele,mxbou,imax,iplot)
implicit real*8(a-h,o-z)
dimension coord(mxpoi,2), alf0(mxpoi,2)
dimension phi0(mxpoi), u0(mxpoi), v0(mxpoi)
integer intma(mxele,nd), iboun(mxbou,4), node(imax,imax)

c
c   Read Input File
c
read(*,*)nx,ny
read(*,*)xmin,xmax,ymin,ymax

```

```

        read(*,*)dt,time_final,iplot
c
c    check bounds
c
    if (nx*ny.gt.mxpai) then
        write(*, '(" Error! - Need to Enlarge MXPAI")')
        stop
    else if ((nx-1)*(ny-1).gt.mxele) then
        write(*, '(" Error! - Need to Enlarge MXELE")')
        stop
    else if (2*(nx-1)+2*(ny-1).gt.mxbou) then
        write(*, '(" Error! - Need to Enlarge MXBOU")')
        stop
    endif
c
c    set some constants
c
    pi=4.0*atan(1.0)
    ntime=nint(time_final/dt)
    dt=dt*2.0*pi
    time_final=time_final*2.0*pi
    xm=0.5*(xmax+xmin)
    ym=0.5*(ymax+ymin)
    dx=(xmax-xmin)/(nx-1)
    dy=(ymax-ymin)/(ny-1)
    xl=xmax-xmin
    yl=ymax-ymin
    w=1.0
    cx=0.25*xl
    cy=0.50*yl
    h=100.0
    rc=0.125*xl
    velmax=-1e5
c
c    set the Initial Conditions
c
    ip=0
    do j=1,ny
        y=ymin + real(j-1)*dy
        do i=1,nx
            x=xmin + real(i-1)*dx
            ip=ip+1
            r=sqrt( (x-cx)**2 + (y-cy)**2 )
            phi0(ip)=0.0

```

```

        if (r.lt.rc) then
            phi0(ip)=h*(1.0-r/rc)
        endif
        u0(ip)=+(y-ym)
        v0(ip)=- (x-xm)
        vel1=u0(ip)**2 + v0(ip)**2
        velmax=max(velmax,vel1)
    end do
end do
cfl=dt*sqrt(velmax/(dx**2 + dy**2))
print*, ' ** CFL = ', cfl
c
c  GENERATE COORD
c
npoin=nx*ny
ip=0
do j=1,ny
    do i=1,nx
        ip=ip+1
        node(i,j)=ip
        coord(ip,1)=xi + dx*real(i-1)
        coord(ip,2)=yi + dy*real(j-1)
        alf0(ip,1)=0.5*dx
        alf0(ip,2)=0.5*dy
    end do
end do
c
c  GENERATE INTMA
c
nelem=(nx-1)*(ny-1)
ie=0
do j=1,ny-1
    do i=1,nx-1
        ie=ie+1
        intma(ie,1)=node(i,j)
        intma(ie,2)=node(i+1,j)
        intma(ie,3)=node(i+1,j+1)
        intma(ie,4)=node(i,j+1)
    end do
end do
c
c  GENERATE IBOUN
c
nboun=2*(nx-1)

```

```

        ib=0

c      bottom (y=ymin)
        ie=1
        do i=1,nx-1
            ib=ib+1
            iboun(ib,1)=node(i,1)
            iboun(ib,2)=node(i+1,1)
            iboun(ib,3)=ie
            iboun(ib,4)=3
            ie=ie+1
        end do

c      top (y=ymax)
        ielem=nelem
        do i=nx,2,-1
            ib=ib+1
            iboun(ib,1)=node(i,ny)
            iboun(ib,2)=node(i-1,ny)
            iboun(ib,3)=ielem
            iboun(ib,4)=3
            ielem=ielem-1
        end do

        return
    end

*-----*
*This subroutine interpolates the right hand side function for the
*2D Advection Equation using a 3-time level Semi-Lagrangian
*Semi-Implicit Method
*Written by F.X. Giraldo on 2/95
*-----*

        subroutine interp(f,phim,u0_x,v0_y,dphim,du0_x,dv0_y,coord,
$                      alf0,node,npoin,dt,xmin,xmax,ymin,ymax,ym,
$                      nx,ny,nd,mxpoi,imax)
        implicit real*8(a-h,o-z)
        dimension f(mxpoi)
        dimension phim(mxpoi), u0_x(mxpoi), v0_y(mxpoi)
        dimension dphim(mxpoi),du0_x(mxpoi),dv0_y(mxpoi)
        dimension coord(mxpoi,2), alf0(mxpoi,2)
        integer node(imax,imax)

        do ip=1,npoin
c

```



```

c      1st, Interpolate "-" values = F( x - 2*alpha, t - dt )
c
      xd=coord(ip,1) - 2.0*alf0(ip,1)
      yd=coord(ip,2) - 2.0*alf0(ip,2)
      if (xd.lt.xmin) xd=xmax - (xmin - xd)
      if (xd.gt.xmax) xd=xmin + (xd - xmax)
      if (yd.lt.ymin) yd=ymax - (ymin - yd)
      if (yd.gt.ymax) yd=ymin + (yd - ymax)

      if ( (xd.lt.xmin.or.xd.gt.xmax) .or.
$        (yd.lt.ymin.or.yd.gt.ymax) ) then
        print*, ' Error in INTERP '
        print*, 'XD out of Range = ',xd,yd
        stop
      endif

c      Do Interpolation
      call splin2(phi,coord,phim,dphim,xd,yd,nx,ny,node,mxpoi,imax)
      call splin2(u_x,coord,u0_x,du0_x,xd,yd,nx,ny,node,mxpoi,imax)
      call splin2(v_y,coord,v0_y,dv0_y,xd,yd,nx,ny,node,mxpoi,imax)

      f(ip)=(1.0-dt*(u_x + v_y))/(1.0+dt*(u0_x(ip) + v0_y(ip)))*phi
      end do

      return
      end

*-----*
*This subroutine solves the 2D Advection Equation
*using the Backward Euler with a predictor-corrector strategy
*Written by F.X. Giraldo on 2/95
*-----*

      subroutine matsuno(phim,phi0,phip,um,u0,up,
$                        vm,v0,vp,coord,intma,npoin,nelem,dt,dx,dy,
$                        node,nx,ny,ym,nd,mxpoi,mxele,imax)
      implicit real*8(a-h,o-z)
      dimension phim(mxpoi), phi0(mxpoi), phip(mxpoi)
      dimension um(mxpoi), u0(mxpoi), up(mxpoi)
      dimension vm(mxpoi), v0(mxpoi), vp(mxpoi)
      dimension coord(mxpoi,2)
      integer intma(mxele,nd), node(imax,imax)

c
c      Loop through the points and integrate using Forward Time
c      and Centered Space...
c

```

```

*      Predictor Stage (forward Euler)

do j=1,ny
  j1=j-1
  j2=j+1
  if (j1.lt.1) j1=ny-1
  if (j2.gt.ny) j2=2
  do i=1,nx
    i1=i-1
    i2=i+1
    if (i1.lt.1) i1=nx-1
    if (i2.gt.nx) i2=2
c
c      Set up the nodes in X and Y
c
    ip=node(i,j)
    ip1=node(i1,j)
    ip2=node(i2,j)
    jp1=node(i,j1)
    jp2=node(i,j2)
c
c      integrate PHI
c
    phim(ip)=phi0(ip)
$    -0.5*dt/dx*u0(ip)*( phi0(ip2)-phi0(ip1) )
$    -0.5*dt/dy*v0(ip)*( phi0(jp2)-phi0(jp1) )
c
c      integrate U
c
    um(ip)=u0(ip)
c
c      integrate V
c
    vm(ip)=v0(ip)
  end do

end do

*      Corrector Stage (backward Euler)

do j=1,ny
  j1=j-1
  j2=j+1

```

```

        if (j1.lt.1) j1=ny-1
        if (j2.gt.ny) j2=2
        do i=1,nx
            i1=i-1
            i2=i+1
            if (i1.lt.1) i1=nx-1
            if (i2.gt.nx) i2=2
c
c            Set up the nodes in X and Y
c
            ip=node(i,j)
            ip1=node(i1,j)
            ip2=node(i2,j)
            jp1=node(i,j1)
            jp2=node(i,j2)
c
c            integrate PHI
c
            phip(ip)=phi0(ip)
$      -0.5*dt/dx*um(ip)*( phim(ip2)-phim(ip1) )
$      -0.5*dt/dy*vm(ip)*( phim(jp2)-phim(jp1) )
c
c            integrate U
c
            up(ip)=u0(ip)
c
c            integrate V
c
            vp(ip)=v0(ip)
        end do
c
c        end do
c
c        Apply the Periodic Boundary Conditions
c
        do j=1,ny
            i1=node(1,j)
            i2=node(nx,j)
            phip(i1)=phip(i2)
            up(i1)=up(i2)
            vp(i1)=vp(i2)
        end do
c
        do i=1,nx

```

```

        i1=node(i,1)
        i2=node(i,ny)
        phip(i1)=phip(i2)
        up(i1)=up(i2)
        vp(i1)=vp(i2)
    end do

1000  continue

    return
end

*-----*
*This subroutine writes the output.  It is currently set only to
*print the concentration (or color) function at each node point.
*Written by F.X. Giraldo on 2/95
*-----*

    subroutine output(phi,u,v,npoin,time,nx,ny,mxpoi)
    implicit real*8(a-h,o-z)
    dimension phi(mxpoi), u(mxpoi), v(mxpoi)

    pi=4.0*atan(1.0)
    write(1,'(2(i6,1x),e16.8)') ,nx,ny,time/(2.0*pi)
    write(1,'(e12.4)')(phi(ip), ip=1,npoin)
    return
    end

*-----*
*These next 4 subroutines construct the Hermitian Interpolation functions
*required by the semi-Lagrangian method.
*Obtained from Numerical Recipes.
*Written by F.X. Giraldo on 2/95
*-----*

    subroutine splie2(df,f,coord,nx,ny,node,mxpoi,imax)
    implicit real*8(a-h,o-z)
    parameter (nmax=3000)

c
c    global arrays
c
    dimension coord(mxpoi,2), f(mxpoi), df(mxpoi)
    integer node(imax,imax)

c
c    local arrays
c
    dimension x(nmax), y(nmax), y2(nmax)

```

```

do j=1,ny
  do i=1,nx
    y(i)=f(node(i,j))
    x(i)=coord(node(i,j),1)
  end do
  call spline(y2,y,x,nx,idum,nmax)
  do i=1,nx
    df(node(i,j))=y2(i)
  end do
end do

return
end
*-----*
subroutine splin2(fd,coord,f,df,xd,yd,nx,ny,node,mxpoi,imax)
implicit real*8(a-h,o-z)
parameter (nmax=3000)
c
c  global arrays
c
dimension coord(mxpoi,2), f(mxpoi), df(mxpoi)
integer node(imax,imax)
c
c  local arrays
c
dimension x(nmax), y(nmax), y2(nmax), y22(nmax)

do j=1,ny
  do i=1,nx
    y(i)=f(node(i,j))
    y2(i)=df(node(i,j))
    x(i)=coord(node(i,j),1)
  end do
  call splint(fd,x,y,y2,xd,nx,nmax)
  y22(j)=fd
end do
do j=1,ny
  x(j)=coord(node(1,j),2)
  y(j)=y22(j)
end do
call spline(y2,y,x,ny,idum,nmax)
call splint(fd,x,y,y2,yd,nx,nmax)

return

```

```

end
*-----*
subroutine spline(y2,y,x,n,idum,mx)
implicit real*8(a-h,o-z)
parameter (nmax=3000)
dimension y(mx), y2(mx), x(mx), u(nmax)

if (mx.gt.nmax) then
  write(*, '(" Must expand NMAX in Subroutine Spline")')
  stop
endif

y2(1)=0.0
u(1)=0.0
do i=2,n-1
  sig=(x(i)-x(i-1))/(x(i+1)-x(i-1))
  p=sig*y2(i-1) + 2.0
  y2(i)=(sig - 1.0)/p
  u(i)=(6.0*((y(i+1)-y(i))/(x(i+1)-x(i)) - (y(i)-y(i-1))
$      /(x(i)-x(i-1)))/(x(i+1)-x(i-1))-sig*u(i-1))/p
end do
y2(n)=0.0
do i=n-1,1,-1
  y2(i)=y2(i)*y2(i+1) + u(i)
end do

return
end
*-----*
subroutine splint(yd,x,y,y2,xd,n,mx)
implicit real*8(a-h,o-z)
dimension x(mx), y(mx), y2(mx)

i1=1
i2=n
10 if (i2-i1.gt.1) then
  im=(i2+i1)/2
  if (xd.gt.x(im)) then
    i1=im
  else
    i2=im
  endif
  goto 10
endif
endif

```

```

      x1=x(i1)
      x2=x(i2)
      dx=x2-x1
      a=(x2-xd)/dx
      b=(xd-x1)/dx
      yd=a*y(i1) + b*y(i2)
$    + ((a**3-a)*y2(i1) + (b**3-b)*y2(i2))*(dx**2)/6.0
      return
      end
*-----*
*This subroutine updates the arrays PHIM,UM,VM,ALFM, PHI0,U0,V0,ALFO
*Written by F.X. Giraldo on 2.95
*-----*
      subroutine update(phim,phi0,phip,um,u0,up,vm,v0,vp,alfm,alf0,
$          npoin,mxpoi)
      implicit real*8(a-h,o-z)
      dimension phim(mxpoi), phi0(mxpoi), phip(mxpoi)
      dimension um(mxpoi),  u0(mxpoi),  up(mxpoi)
      dimension vm(mxpoi),  v0(mxpoi),  vp(mxpoi)
      dimension alfm(mxpoi,2), alf0(mxpoi,2)

c
c      Loop through all the nodes and update
c
      do ip=1,npoin
c
c      Update F(x-2*alpha,t-dt)=F(x-alpha,t)
c
          phim(ip)=phi0(ip)
          um(ip)=u0(ip)
          vm(ip)=v0(ip)
          alfm(ip,1)=alf0(ip,1)
          alfm(ip,2)=alf0(ip,2)
c
c      Update F(x-alpha,t)=F(x,t+dt)
c
          phi0(ip)=phip(ip)
          u0(ip)=up(ip)
          v0(ip)=vp(ip)
      end do
      return
      end

```

A Matlab Program to plot the output showing the cone at various time is given

```
clg
i=0;
j=0;
fid=fopen('matlab.out','r');
ab=fscanf(fid,'%d',1);
isets=ab(1)
while (i < isets)
ab=fscanf(fid,'%d%d%f',3);
nx=ab(1)
ny=ab(2)
hour=ab(3)
count=nx*ny;
jj=1;
while (j < ny)
[aa,count]=fscanf(fid,'%e%e%e%e',nx);
ab(jj:jj+count-1)=aa;
j=j+1;
jj=jj+count;
end;
u=reshape(ab,nx,ny);
v=u';
c=contour(v,10);
clabel(c);
    title(['concentration after ',num2str(hour),' revolutions'])
    print c -dps -append
i=i+1;
j=0;
end;
```


Conclusion

We have developed a bilinear finite element Fortran code to solve the two dimensional advection equation on a unix-based SUN Sparc 10 workstation. The stability of the method is analyzed. We have also developed a semi-Lagrangian finite element code. These codes were experimented with in solving the cone test problem. It is clear from the plots that the semi-Lagrangian is superior to the Eulerian finite elements, since the cone rotates back to its position without leaving a noisy trail behind.

Acknowledgement

The first author would like to thank the National Research Council and the Naval Postgraduate School for their support. The second author would like to thank Dr. Zahari Zlatev at the Danish Environmental Research Institute for his support during his two week visit to the Institute. This research was supported in part by the Naval Postgraduate School.

Appendix

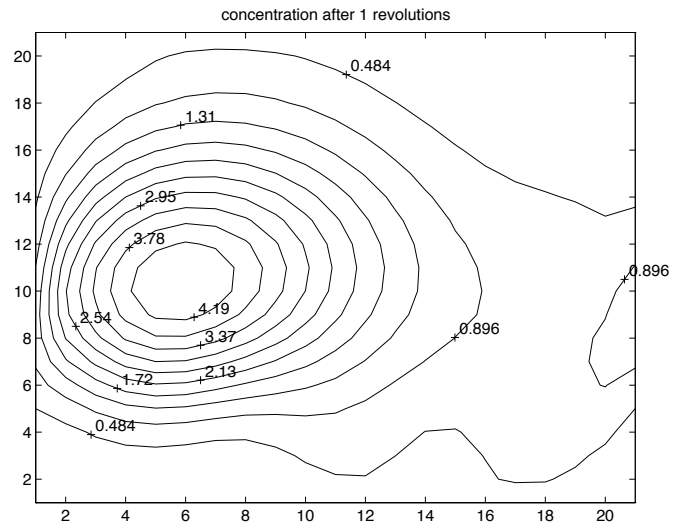


Figure (1) Eulerian FEM explicit time integration

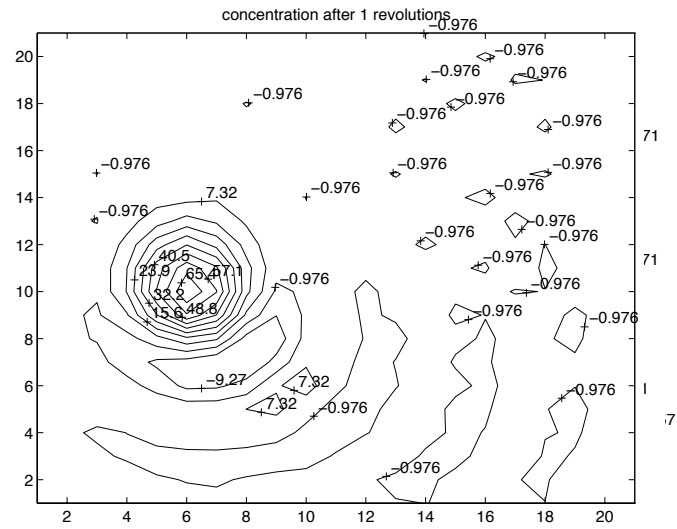
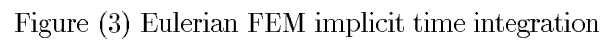


Figure (2) Eulerian FEM Crank-Nicholson time integration



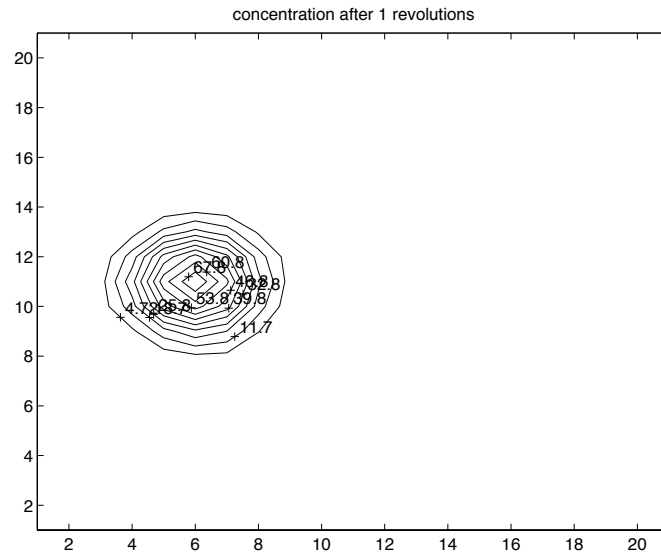


Figure (4) Semi-Lagrangian FEM

References

1. Z. Zlatev, Numerical Treatment of Large Air Pollution Models, Kluwer Acad. Pub., Dordrecht, 1995.
2. R. D. Richtmeyer and K. W. Morton, Difference Methods for Initial Value Problems, Interscience Pub., New York, 1967.
3. S. A. Orszag, Transform method for calculation of vector-coupled sums: Application to the spectral form of the vorticity equation, *J. Atmos. Sci.*, **27** (1970), 890-895.
4. S. A. Orszag, Comparison of pseudo spectral and spectral approximations, *Stud. Appl. Math.*, **51** (1972), 75-112.
5. B. Neta and R. T. Williams, Stability and phase speed for various finite element formulations of the advection equation, *Computers and Fluids*, **14** (1986), 393-410.
6. D. W. Pepper, C. D. Kern and P. E. Long Jr., Modelling the dispersion of atmospheric pollution using cubic splines and chapeau functions, *Atmos. Environ.*, **13** (1979), 223-237.
7. B. Neta, Analysis of Finite Elements and Finite Differences for Shallow Water Equations: A Review, *Mathematics and Computers in Simulation*, **34** (1992), 141-162.
8. A. Staniforth and C. Beaudoin, On the efficient evaluation of certain integrals in the Galerkin F. E. method, *Intern. J. Numer. Meth. Fluids*, **6** (1986), 317-324.
9. A. Staniforth and J. Côté, Semi-Lagrangian integration schemes for atmospheric models - a review, *Monthly Weather Review*, **119** (1991), 2206-2223.
10. A. L. Schoenstadt, The effect of spatial discretization on the steady state and transient solutions of a dispersive wave equation, *J. Comp. Phys.*, **23**, (1977), 364-379.
11. R. Peyret and T. D. Taylor, Computational Methods for Fluid Flow, Springer Verlag, New York, 1986.
12. G. J. Haltiner, R. T. Williams, Numerical Prediction and Dynamic Meteorology, Wiley, New York, 1980.
13. J. M. Ortega, Introduction to Parallel and Vector Solutions of Linear Systems, Plenum Press, New York, 1988.